Astrée: Nachweis der Abwesenheit von Laufzeitfehlern

Daniel Kästner[1], Christian Ferdinand[1], Stephan Wilhelm[1], Stefana Nenova[1],
Olha Honcharova[1], Patrick Cousot[2], Radhia Cousot[2], Jérôme Feret[2],
Laurent Mauborgne[2], Antoine Miné[2], Xavier Rival[2], Élodie-Jane Sims[2]

[1] AbsInt GmbH, Saarbrücken, Germany
[2] École Normale Supérieure, Paris, France

05.06.2009

Kurzzusammenfassung

Sicherheitskritische eingebettete Systeme müssen hohen Qualitätsanforderungen genügen. Laufzeitfehler, z.B. arithmetische Überläufe oder Rundungsfehler können zu fehlerhaftem Programmverhalten führen. Da in der Regel keine vollständige Testabdeckung möglich ist, bieten sich statische Analysatoren an. Diese bieten eine vollständige Coverage, können jedoch Fehlalarme erzeugen. Da jeder potentielle Laufzeitfehler manuell vom Benutzer überprüft werden muss, kann eine hohe Zahl von Fehlalarmen dazu führen, dass echte Fehler übersehen werden. Der statische Analysator Astrée kann durch Spezialisierung und Parametrisierung an die zu analysierende Software angepasst werden. Dies ermöglicht kurze Analysezeiten und eine niedrige Zahl von Fehlalarmen. Astrée wird z.B. bei der Zertifizierung von industrieller Flugzeugsteuerungssoftware eingesetzt.

Schlüsselwörter

Softwarevalidierung, Laufzeitfehler, statische Analyse.

1 Einführung

Die Test- und Validierungsphase stellt einen erheblichen Kostenfaktor bei der Entwicklung sicherheitskritischer Systeme dar. Unentdeckte Fehler können schwere Schäden verursachen. Daher stellt sich die Frage, wie mit vertretbarem Aufwand das korrekte Funktionieren der Software nachgewiesen werden kann. In der Luftfahrt- und Automobilindustrie werden zunehmend statische Analysatoren auf der Basis der Abstrakten Interpretation eingesetzt, um Programmeigenschaften sicherheitskritischer Software zu validieren. Beispiele hierfür sind der Nachweis der längstmöglichen Ausführungszeit [17] und des maximalen Stackverbrauchs von Tasks [10], Codeinspektion [14], die Bestimmung der Genauigkeit von Gleitkommaberechnungen [11] und der Ausschluss des Auftretens von Laufzeitfehlern [2, 6, 8]. Moderne statische Analysatoren skalieren gut und erlauben somit die Analyse vollständiger sicherheitskritischer Industrieanwendungen.

Der vorliegende Artikel konzentriert sich auf eine bestimmte Fehlerklasse, nämlich die sogenannten *Laufzeitfehler*. Laufzeitfehler führen in der Regel dazu, dass nach ihrem Auftreten das Programmverhalten undefiniert ist. Einige Laufzeitfehler, z.B. floating-point Überläufe, können Interrupts auslösen und somit abgefangen werden – vorausgesetzt, ein entsprechender Interrupthandler wurde implementiert. Die möglichen Folgen von Laufzeitfehlern wie beispielsweise Feldgrenzenverletzungen beim Zugriff auf Feldelemente (`array access out-of-bounds`) oder die Dereferenzierung eines ungültigen Zeigers reichen von fehlerhaftem Programmverhalten bis hin zum Softwareabsturz.

Ein wichtiges Ziel bei der Entwicklung sicherheitskritischer Software ist daher der Nachweis, dass keine solchen Fehler zur Laufzeit auftreten können. Ein Nachweis der Abwesenheit von Laufzeitfehlern kann durch Softwaretests nicht erbracht werden, da in der Regel keine vollständige Testabdeckung möglich ist. Mit Hilfe statischer Analyse kann ein solcher Nachweis selbst für große Softwareprojekte geführt werden. Ein Grund für den Erfolg statischer Analysetechniken liegt darin, dass sie auf einer *sicheren Überapproximation* der konkreten Programmsemantik basieren. Das bedeutet, dass das Ergebnis einer solchen Analyse für eine Anweisung x entweder "x wird niemals zu einem Fehler führen" oder "x könnte einen Fehler verursachen" lautet. Im ersten Fall ist die Fehlerfreiheit garantiert, im zweiten Fall liegt entweder ein Fehler vor, oder es handelt sich um einen Fehlalarm. Diese Unschärfe ermöglicht es statischen Analysatoren, selbst für große Softwareprojekte innerhalb akzeptabler Zeit ein Ergebnis zu berechnen. Die Analyseergebnisse sind trotzdem verlässlich, da sich der Analysator bei korrektem Design nur auf der sicheren Seite irren kann: Wenn die Analyse keinen Fehler findet, ist der Nachweis der Fehlerfreiheit geführt. Die Abdeckung liegt bei 100%.

Es stellen sich somit zwei zentrale Anforderungen an einen Analysator: Da jede Fehlermeldung überprüft werden muss, ist es zum einen wichtig, dass der Analysator präzise ist, d.h. nur wenige Fehlalar-

me produziert. Das ist nur möglich, wenn der Analysator für die jeweilige Programmfamilie und eine bestimmte Klasse von Programmeigenschaften spezialisiert werden kann. Zum anderen muss der Analysator so parametrisierbar sein, dass Benutzer eine Feinabstimmung der Analyse für jedes beliebige Programm dieser Klasse vornehmen können. Nur auf diese Weise kann das Ziel "null Fehlalarme" erreicht werden. Verallgemeinernde Softwarewerkzeuge liefern oft eine hohe Anzahl falscher Alarme, weshalb derartige Tools in der Industrie oft nur zur Fehlersuche, aber nicht zum Nachweis der Abwesenheit von Laufzeitfehlern eingesetzt werden.

Schwerpunkt unseres Artikels ist der statische Analysator Astrée (*Analyseur statique de logiciels temps-réel embarqués*) [1], der an der École Normale Supérieure entstand [9] und von AbsInt unter Lizenz der CNRS/ENS weiterentwickelt und vertrieben wird.

Astrée wurde gezielt mit Blick auf die oben genannten Anforderungen entwickelt: Eingesetzt zum Nachweis der Abwesenheit von Laufzeitfehlern erzeugt es nur wenige Fehlalarme und bietet Benutzern vielfältige Möglichkeiten zur weiteren Reduktion der Zahl von Fehlalarmen. Im Unterschied zu vielen anderen statischen Analysatoren kann Astrée somit nicht nur zum Finden von Laufzeitfehlern verwendet werden, sondern auch tatsächlich zum Nachweis der Abwesenheit von Laufzeitfehlern.

Bei Airbus wurde Astrée erfolgreich zur Analyse von sicherheitskritischer Avioniksoftware eingesetzt [16]. Im vorliegenden Artikel geben wir einen Überblick über die Struktur von Astrée und beschreiben, wie Entwickler die Abwesenheit von Laufzeitfehlern nachweisen und dabei die Anzahl an Fehlalarmen und somit den erforderlichen Aufwand minimieren können.

Wir beschreiben zunächst informell die Grundlagen der statischen Programmanalyse (Kap. 2). Kap. 3 gibt einen Überblick über das Design von Astrée, beschreibt das Paradigma *Null Fehlalarme* und demonstriert die Handhabung von Astrée anhand eines Beispiels. Zum Erreichen einer hohen Analysepräzision ist es wichtig, Astrée für die jeweilige Programmfamilie zu spezialisieren und die Analyseparameter an das konkrete Zielprogramm anzupassen. Kap. 4 geht auf die wichtigsten Analyse-Domains von Astrée ein und beschreibt exemplarisch einige Optionen und Direktiven, mit denen die Analyse parametrisiert werden kann. In Kap. 5 werden experimentelle Daten über den Einsatz von Astrée auf relevanten Softwareprojekten beschrieben und Kap. 6 gibt einen zusammenfassenden Ausblick.

2 Abstrakte Interpretation

Statische Datenflussanalysen berechnen Invarianten für alle Programmpunkte durch Fixpunktiteration über die Programmstruktur oder den Kontrollflussgraphen. Die Theorie der Abstrakten Interpretation [5] bietet eine semantikbasierte Methodik zur Durchführung von statischen Programmanalysen. Mit Hilfe von Abstraktionsfunktionen wird die konkrete Semantik auf eine abstrakte Semantik abgebildet. Während die meisten interessanten Programmeigenschaften in der konkreten Semantik unentscheidbar sind, kann die abstrakte Semantik so gewählt werden, dass sie berechenbar ist. Die statische Analyse wird dann bezüglich der abstrakten Semantik durchgeführt. Durch Konkretisierungsfunktionen wird die abstrakte Semantik wieder auf die konkrete Semantik abgebildet. Mit Hilfe der Abstrakten Interpretation läßt sich beweisen, dass die Analyse terminiert und die konkrete Semantik überapproximiert, die Analyseergebnisse also verlässlich (*sound*) sind. Für eine *verlässliche* Laufzeitfehleranalyse muss beispielsweise die folgende Eigenschaft gelten: Besagt das Analyseergebnis, dass kein Laufzeitfehler auftreten kann, dann kann es keine konkrete Programmausführung geben, bei der ein Fehler auftritt. Hier ist das Ergebnis also: *Es tritt definitiv kein Fehler auf.*

Meldet die Analyse einen möglichen Laufzeitfehler, kann es sich aufgrund der Analyseungenauigkeit um einen Fehlalarm handeln. In diesem Fall gibt es keine konkrete Programmausführung, bei der der Fehler tatsächlich auftritt. Ein Alarm bedeutet also: *Es handelt sich möglicherweise um einen Fehler.* Ungenauigkeiten können also nur auf der sicheren Seite auftreten.

Eine statische Programmanalyse kann auf Basis der *Abstrakten Interpretation* formal spezifiziert werden. Aus einer solchen Spezifikation kann mit Hilfe des Programmanalysatorgenerators PAG [12] die Implementierung des Analysators automatisch generiert werden. Dadurch werden das Risiko von Implementierungsfehlern und gleichzeitig die Entwicklungszeit deutlich gesenkt. Die Spezifikation besteht aus den folgenden Komponenten:

- Abstrakter Domain: enthält die abstrakten Werte, in denen die berechneten Invarianten ausgedrückt werden.
- Transferfunktionen: beschreiben die möglichen Veränderungen der abstrakten Werte an jedem Programmpunkt.
- Gleichheitstest: zum Überprüfen eines Fixpunkts der Transferfunktion.
- *Least Upper Bound Operator*: zur Kombination von Informationen an Kontrollflussverschmelzungen. Dieser Operator ist in der Regel mit einem Informationsverlust verbunden.

Für eine korrekte Analysedefinition müssen diese Komponenten eine Reihe von Bedingungen erfüllen. Der abstrakte Domain muß ein vollständiger Verband

sein, der die aufsteigende Kettenbedingung erfüllt, und die Transferfunktionen müssen in der Regel monoton sein. Für weitergehende Informationen sei auf [15, 4] verwiesen.

Anhand der konkreten Semantik sind die meisten interessanten Programmeigenschaften unentscheidbar. Durch Reduktion der Präzision kann die Berechnung beschleunigt werden. Die abstrakten Domains müssen hierzu geschickt definiert werden.

3 Astrée — Design und Überblick

Astrée [2] ist ein parametrisierbarer statischer Analysator auf der Basis der Abstrakten Interpretation zum Nachweis der Abwesenheit von Laufzeitfehlern in C-Programmen nach "ISO/IEC 9899:1999 (E)" (C99-Standard) [3]. Astrée ist einsetzbar für strukturierte C-Programme mit komplexem Speicherverhalten, jedoch ohne dynamische Speicherallokation oder unbeschränkte Rekursionen. Hierzu zählen viele eingebettete Steuerungsprogramme im Bereich von Kernenergie, Medizintechnik, Automobilelektronik sowie Luft- und Raumfahrt, insbesondere synchrone Steuerungssoftware wie z.B. Flugsteuerungssoftware.

Astrée deckt unter anderem die folgenden Fehlersituationen ab: Verletzung von Feldgrenzen, ganzzahlige Division durch Null, ungültige Pointerzugriffe, Überläufe und ungültige Operationen auf Gleitkommazahlen, Wrap-arounds bei ganzzahliger Arithmetik, sowie Typkonversionen, die zu Wrap-arounds führen. Außerdem kann Astrée Aussagen über beliebige benutzerdefinierte C-Assertions im Programm berechnen.

Die Definition einer formalen Semantik für C Programme als Grundlage eines statischen Analysators ist aufgrund des gewährten Implementierungsspielraums schwierig. So gibt der "ISO/IEC 9899:1999 (E)" Standard beispielsweise an, dass nach einem *Laufzeitfehler* das Verhalten undefiniert ist. Ein Schreibzugriff über einen ungültigen Zeiger oder ein Schreibzugriff in ein Array mit ungültigem Index können den gesamten Speicher korrumpieren. Das Resultat einer solchen Programmausführung kann statisch nicht vorhergesagt werden. Astrée unterscheidet daher zwischen zwei verschiedenen Typen von Laufzeitfehlern [6]:

1.) Bei Laufzeitfehlern, die zu *undefiniertem* Verhalten führen, gibt Astrée einen Alarm aus und berücksichtigt für die weitere Analyse nur Szenarien, in denen der Fehler nicht auftritt. Beispiele für diese Kategorie sind ganzzahlige Divisionen durch 0, Gleitkomma-Überläufe, ungültige Feldzugriffe, oder Zugriffe auf ungültige Zeiger. Ein Beispiel ist das folgende Programm:

```
#include <stdio.h>
int main() {
  int n, T[0];
  n = 2147483647;
  printf("n=%i, T[n]=%i\n", n, T[n]);
}
```

Dieses Programm führt auf unterschiedlichen Rechnern zu unterschiedlichen Ausgaben:
`n=2147483647, T[n]=2147483647` (PPC Mac)
`n=2147483647, T[n]=-1208492044` (Intel Mac)
`n=2147483647, T[n]=-135294988` (32-bit PC Intel)
`Bus error` (64-bit PC Intel)

Da unvorhersehbar ist, was nach einem solchen Fehler passiert, unternimmt Astrée nicht den Versuch einer Vorhersage. Der Analysator nimmt statt dessen an, dass die Programmausführung im Fehlerfall stoppt und berücksichtigt im weiteren Verlauf der Analyse nur Szenarien, in denen der Fehler nicht auftritt. Tritt ein Fehler in einem bestimmten Ausführungskontext definitiv auf, meldet Astrée einen definitiven Fehler und beendet die Analyse für diesen Kontext.

2.) Bei Laufzeitfehlern, die zu *unspezifiziertem*, aber *vorhersehbarem* Verhalten führen, gibt Astrée einen Alarm aus und berücksichtigt alle möglichen Ergebnisse im weiteren Verlauf der Analyse. Beispiele hierfür sind ganzzahlige Überläufe oder ungültige Shiftoperationen, bei denen die tatsächlichen Berechnungen vom erwarteten mathematischen Resultat abweichen.

3.1 Null Fehlalarme

Ein wichtiges Ziel für den Industrieeinsatz ist das Erreichen von *null Fehlalarmen*. Das Ziel von Astrée ist nicht, nach Laufzeitfehlern zu suchen, sondern zu beweisen, dass es niemals zu einem Laufzeitfehler kommen wird. Der Beweis über die Abwesenheit von Laufzeitfehlern ist nur geführt, wenn die Analyse ohne jeden Alarm terminiert.

Daher sollte grundsätzlich jeder Alarm vom Entwickler überprüft werden. Liegt ein echter Fehler vor, sollte er behoben werden und die Analyse erneut gestartet werden; handelt es sich um einen Fehlalarm, kann dieser möglicherweise durch geeignete Parametrisierung des Analysators vermieden werden (siehe Kap. 4). Geht der Entwickler davon aus, dass der Fehlerfall aufgrund bestimmter Voraussetzungen nicht auftreten kann, die Astrée nicht bekannt sind, besteht die Möglichkeit, diese dem Analysator explizit durch Annotationen bekannt zu geben (vgl. Kap. 4.2). Kann Astrée statisch eine Verletzung solcher Annahmen feststellen, wird ein Alarm erzeugt. Ansonsten liegt mit den Annotationen eine explizite Spezifikation von Bedingungen vor, die für eine korrekte Programmausführung erfüllt sein müssen.

Von zentraler Bedeutung ist es daher, Entwicklern möglichst viele Informationen über Entstehung und Ursache eines Alarms zur Verfügung zu stellen. Selbstverständlich ist auch eine hohe Analysepräzision wichtig, um möglichst wenig Fehlalarme auszulösen.

3.2 Alarmanalyse

Betrachten wir das folgende Eingabeprogramm:

```
 1:  #define BASE 0x80000000
 2:  #define OFFSET 0x38343031
 3:  volatile int SwitchPosition;
 4:
 5:  int main()
 6:  {
 7:    /*...*/
 8:    int MODULE1 = BASE + OFFSET;
 9:    /*...*/
10:    char sp = SwitchPosition;
11:  }
```

Astrée meldet für dieses Programm zwei potentielle Laufzeitfehler. Die Zeilenangaben beziehen sich dabei jeweils auf den präprozessierten Code, der in der GUI zusammen mit dem ursprünglichen Sourcecode angezeigt wird (siehe Abb. 1).

```
bsp1.c:11.22-45::[call#main@8:]:
  ALARM: implicit unsigned int->signed int
  conversion range {3090427953} not included
  in [-2147483648, 2147483647]
bsp1.c:13.12-26::[call#main@8:]:
  ALARM: implicit signed int->unsigned char
  conversion range [-2147483648, 2147483647]
  not included in [0, 255]
```

Die Meldung zeigt die genaue Codestelle an, wo der Fehler auftritt (Datei `bsp1.c` nach Präprozessierung, Zeile 11 — bzw. Zeile 8 im Originalcode —, Zeichen 22-45) und gibt den Aufrufkontext an, in dem dieser Programmpunkt erreicht wird. Im Beispiel ist dies einfach der Aufruf von Funktion `main`, die in Zeile 8 (Zeile 5 im Originalcode) beginnt. Durch Mausklick auf den Alarm gelangt man direkt zu den jeweiligen Codestellen; der potentiell fehlerhafte Ausdruck ist dabei markiert. Desweiteren wird nicht analysierter Code von Astrée markiert. Dies erleichtert die Untersuchung von Alarmen und ermöglicht es, unerreichbaren Code zu identifizieren.

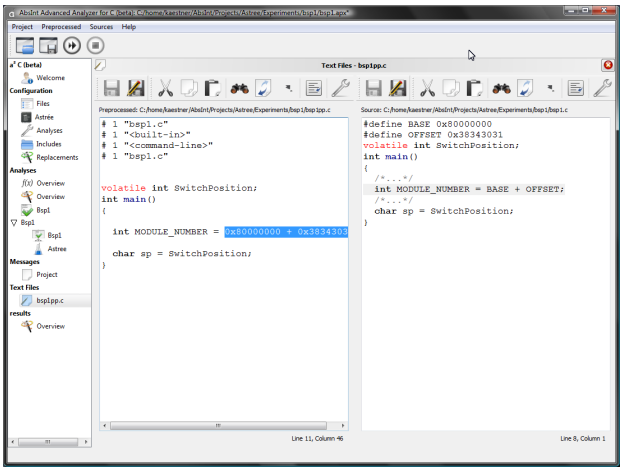

Abbildung 1: Codeansicht von Astrée.

Für jeden Fehler muss der Benutzer analysieren, ob er in einer realen Programmausführung tatsächlich auftreten kann. Der Alarm von Zeile 8 wird von einem echten Laufzeitfehler verursacht. Ursache ist die Behandlung hexadezimaler Konstanten nach dem C99-Standard. Der Typ einer solchen Konstante wird als `int` angenommen, falls er in den `signed int`-Wertebereich passt, ansonsten als `unsigned int`, etc. In diesem Fall wird als Typ `unsigned int` angenommen, da `0x80000000`$= 2^{31}$, was nicht in ein 32-bit `signed int`, aber in ein 32-bit `unsigned int` passt. Da 0x38343031> 0, ist das Resultat der Addition vom Typ `unsigned int` und liegt auserhalb des `signed int`-Wertebereichs. Eine mögliche Problembehebung ist es, `MODULE1` als `unsigned int` zu deklarieren.

Wurde der Code unter der Voraussetzung implementiert, dass es für den modellierten Schalter 8 verschiedene Positionen gibt, handelt es sich bei der zweiten Warnung von Astrée um einen Fehlalarm. Der Wert von SwitchPosition kann sich ändern, allerdings nur Werte aus $0, \ldots, 7$ annehmen. Diese Information kann Astrée durch die Direktive `__ASTREE_volatile_input((SwitchPosition [0,7]))` mitgeteilt werden. Ändert sich das Eingabeprogramm im Laufe der Zeit, ist die Gültigkeit solcher Annotationen immer explizit zu prüfen.

Der Aufruf von Astrée für das geänderte Programm zeigt, dass das Programm nun frei von Laufzeitfehlern ist.

4 Parametrisierung von Astrée

Der C99-Standard legt keine Größen von Datentypen, keine Endianness und keine Alignmentvorgaben fest. Astrée kann mit einer separaten Konfigurationsdatei an unterschiedliche ABIs angepaßt werden und damit die korrekte Maschinenrepräsentation berücksichtigen.

Um eine hohe Präzision zu erzielen, kann Astrée zudem genau an die zu analysierenden Softwareprojekte angepaßt werden. Um dies zu erleichtern, ist Astrée vollständig parametrisch bezüglich des abstrakten Analysedomains. Neue abstrakte Domains können eingebunden werden. Durch Untersuchung auftretender Fehlalarme können existierende abstrakte Domains schrittweise verfeinert und an die Bedürfnisse bestimmter Programmklassen angepasst werden.

4.1 Anpassung an Programmfamilien

Astrée bietet eine Reihe vordefinierter abstrakter Domains. Durch geeignete Auswahl der Domains, die Astrée für eine Analyse berechnen soll, kann Astrée für eine bestimmte Programmklasse spezialisiert werden.

Einige der wichtigsten Astrée-Domains werden im folgenden kurz beschrieben: der *Interval Domain* approximiert Variablenwerte durch Intervalle, der *Octagon Domain* entdeckt Beziehungen der Form $X \pm Y \leq$

c, wobei X und Y Programmvariablen und c eine numerische Konstante ist [2]. Der *Floating-point Domain* dient zur präzisen Modellierung von Gleitkommaberechnungen unter Berücksichtigung möglicher Rundungsfehler. Der *Memory Domain* bildet Variablen auf abstrakte Zellen ab, die einzelnen Speicherzellen oder Mengen von Speicherzellen entsprechen können. Er ermöglicht es Astrée, Pointer-Arithmetik und Manipulationen von Unions präzise zu analysieren. Ein spezieller Domain für *Entscheidungsbäume* erlaubt die präzise Analyse von Programmverzweigungen. Der *Clock Domain* wurde speziell für synchrone Steuerungsprogramme entwickelt und erlaubt, Variablenwerte in Beziehung zum Systemtakt zu setzen. Der *Filter Domain* ermöglicht die präzise Approximation von digitalen Filtern.

Für synchrone Echtzeitprogramme sind der *Clock Domain*, der *Filter Domain*, der *Boolean Tree Domain* und der *Floating-point Domain* wichtig. So kann in unterschiedlichen Anwendungsszenarien die Analyse auf die jeweils benötigten Domains eingeschränkt werden.

4.2 Fine-Tuning

In einem weiteren Schritt ist auch eine gezielte Parametrisierung von Astrée zur Anpassung an ein bestimmtes Programm aus der jeweiligen Programmklasse möglich. Hierzu stehen zwei Arten von Parametern zur Verfügung. Zum einen sind dies Annotationen bzw. Optionen, mit denen der abstrakte Domain parametrisiert werden kann und somit die Präzision der Analyse für bestimmte Programmkonstrukte beeinflußt werden kann. Zum anderen gibt es Annotationen zur Bereitstellung externer Informationen an Astrée in Form von semantischen Hypothesen.

4.2.1 Parametrisierung von abstrakten Domains

Ein wichtiges Beispiel für diesen Parametrisierungstyp ist das *semantische Schleifenabrollen*. Semantisches Schleifenabrollen erlaubt es dem Analysator, zwischen verschiedenen Iterationen einer Schleife zu unterscheiden und eine bessere Analysepräzision zu erreichen. Wird eine Schleife n-mal abgerollt, werden für die ersten n Iterationen eigene Invarianten berechnet, und nur die späteren Iterationen in einer gemeinsamen Schleifeninvariante zusammengefaßt. Im allgemeinen wird die Analyse mit zunehmendem Abrollfaktor präziser, gleichzeitig steigt aber auch die Analysezeit. Semantisches Schleifenabrollen ist eine Spezialanwendung des *Trace Partitioning* Domains von Astrée [13].

Mit Hilfe von *Smashing* bzw. n-fachem *Folding* kann Astrée die Werte vieler Speicherstellen durch eine einzige Zelle, bzw. n Zellen repräsentieren. Speicherbedarf und Laufzeit der Analyse sinken hierdurch, die Analysepräzision verschlechtert sich jedoch. Im einfachsten Fall von Array-Smashing wird zum Beispiel ein Array als eine einzige Arrayzelle abstrahiert. Ab einem bestimmten Schwellwert für die Bytegröße von Datenstrukturen wird automatisches Smashing durchgeführt. Dieser Schwellwert kann global durch eine Kommandozeilenoption gesetzt werden und, ebenso wie der Foldingfaktor n, durch Annotationen lokal für individuelle Datenstrukturen modifiziert werden. Es stehen zahlreiche weitere Optionen zur Verfügung, auf die hier aus Platzgründen nicht näher eingegangen werden kann.

4.2.2 Semantische Hypothesen

Durch die Direktive `__ASTREE_known_fact((B))` kann der Benutzer Astrée Zusatzinformationen zur Verfügung stellen. `B` ist hierbei ein Boolescher Ausdruck in C-Syntax ohne Seiteneffekte. Astrée nimmt dann an, dass am Programmpunkt der Annotation die Bedingung `B` erfüllt ist, ohne diese Hypothese zu überprüfen. Kann Astrée allerdings beweisen, dass `B` immer falsch ist, wird eine Warnung ausgegeben. Ein einfaches Beispiel ist `__ASTREE_known_fact((i>0))`.

Astrée nimmt bei `volatile`-Variablen an, dass sich deren Wert an jedem beliebigen Programmpunkt ändern kann[1]. Standardmäßig nimmt Astrée hierbei an, dass jeder mögliche Wert des Variablentyps angenommen werden kann. Oftmals ist der Wertebereich jedoch beschränkt, z.B. im Kontext von Sensorwerten. Solche Wertebereiche können mit Hilfe der Direktive `__ASTREE_volatile_input` angegeben werden (siehe Kap. 3.2, S. 4).

Die Direktive `__ASTREE_assert((B))` veranlasst Astrée zu überprüfen, ob der Boolesche Ausdruck B an diesem Programmpunkt stets erfüllt ist. Gibt es einen Kontext, in dem B falsch ist, gibt Astrée eine Warnung aus.

5 Praktische Erfahrungen

Astrée wurde auf industrieller Ebene für verschiedene Projekte aus Luft- und Raumfahrt eingesetzt. Eins der untersuchten Softwareprojekte aus dem Avionikbereich umfaßt 132000 Zeilen C-Code mit Makros und enthält etwa 10.000 globale und statische Variablen [2]. Beim ersten Lauf von Astrée traten 1200 Fehlalarme auf; nach Anpassungen von Astrée konnte die Zahl der Fehlalarme auf 11 reduziert werden. Die Analysedauer betrug 1h 50 min auf einem PC (2.4 GHz, 1GB RAM).

In [7] wird der Analyseprozeß für ein Avionikprojekt von Airbus im Detail beschrieben. Das Softwareprojekt besteht aus 200000 Zeilen präprozessierten C-Codes, führt viele Gleitkommaberechnungen aus und

[1] Bei Angabe eines speziellen Kommandozeilenparameters wird das Schlüsselwort volatile von Astrée ignoriert. Hintergrund hierfür ist, dass Variablen in der Praxis oft als volatile deklariert werden, um Compileroptmierungen zu verhindern, es sich also nicht um echte volatiles handelt.

beinhaltet digitale Filter. Die Analysedauer für das Gesamtprogramm beträgt hier etwa 6 Stunden auf einem 2.6 GHz PC mit 16 GB RAM. Der erste Analyselauf ergab 467 Alarme. Durch Programmkorrekturen, Anpassen der Analyseparameter und Einfügen von Analysedirektiven konnte die Zahl der Alarme auf Null reduziert werden.

6 Zusammenfassung und Ausblick

Softwarefehler in sicherheitskritischen eingebetteten Systemen können schwere Schäden verursachen. Der Nachweis der Fehlerfreiheit von Software gewinnt im Rahmen von Entwicklungsnormen wie DO178B oder ISO26262 zunehmend an Bedeutung. Softwarewerkzeuge basierend auf statischer Programmanalyse bieten eine vollständige Coverage und können dazu beitragen, den Testaufwand erheblich zu reduzieren. Wichtig ist es dabei, eine hohe Analysepräzision zu erreichen, um möglichst wenige Fehlalarme zu produzieren. Da Fehlalarme weitere mögliche Laufzeitfehler überdecken können, ist es wichtig, jeden Alarm manuell zu überprüfen. Ein Analysator sollte daher detaillierte Informationen über das Auftreten von Alarmen bereitstellen, um die Fehleruntersuchung zu unterstützen. Desweiteren muss der Analysator so parametrisierbar sein, dass Benutzer eine Feinabstimmung der Analyse für die zu analysierende Software vornehmen und Fehlalarme schrittweise eliminieren können.

Der statische Analysator Astrée wurde gezielt mit Blick auf die oben genannten Anforderungen entwickelt: Eingesetzt zum Nachweis der Abwesenheit von Laufzeitfehlern in C-Programmen erzeugt er nur wenige Fehlalarme und bietet Benutzern vielfältige Möglichkeiten zur weiteren Reduktion der Zahl von Fehlalarmen. Im Unterschied zu vielen anderen statischen Analysatoren kann Astrée somit nicht nur zum Finden von Laufzeitfehlern verwendet werden, sondern auch tatsächlich zum Nachweis der Abwesenheit von Laufzeitfehlern. Industrielle synchrone Echtzeitsoftware aus dem Avionik-Bereich konnte erfolgreich mit Astrée analysiert werden und die Anzahl der Fehlalarme auf Null gesenkt werden.

Literatur

[1] AbsInt GmbH. Astrée Website. http://www.astree.de.

[2] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.

[3] JTC1/SC22. Programming languages – C, 16 Dec. 1999.

[4] P. Cousot. Abstract Interpretation Based Formal Methods and Future Challenges, invited paper. In R. Wilhelm, editor, *¡¡ Informatics — 10 Years Back, 10 Years Ahead ¿¿*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer, 2001.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM Press.

[6] P. Cousot, R. Cousot, J. Feret, A. Miné, L. Mauborgne, D. Monniaux, and X. Rival. Varieties of Static Analyzers: A Comparison with ASTRÉE. In *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, TASE 2007*, pages 3–20. IEEE Computer Society, 2007.

[7] D. Delmas and J. Souyris. ASTRÉE: from Research to Industry. In *Proc. 14th International Static Analysis Symposium (SAS2007)*, number 4634 in LNCS, 2007.

[8] A. Deutsch. Static Verification Of Dynamic Properties. *ACM SIGAda 2003 Conference*, 2003.

[9] École Normale Supérieure. ASTRÉE Website. http://www.astree.ens.fr.

[10] C. Ferdinand, R. Heckmann, and D. Kästner. Static Memory and Timing Analysis of Embedded Systems Code. In *Proceedings of the IET Conference on Embedded Systems at Embedded Systems Show (ESS) 2006, Birmingham*, 2006.

[11] M. Martel. An Overview of Semantics for the Validation of Numerical Programs. *6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI05)*, 2005.

[12] F. Martin. PAG—an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1), 1998.

[13] L. Mauborgne and X. Rival. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *ESOP'05*, 2005.

[14] B. R. Melski, G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. In *In VSTTE*, page 1603, 2005.

[15] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[16] J. Souyris and D. Delmas. Experimental Assessment of Astrée on Safety-Critical Avionics Software. In *SAFECOMP*, pages 479–490, 2007.

[17] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics Software Systems. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003)*, pages 625–632. IEEE Computer Society, June 2003.